Distributed Iterative Quantization for Large Scale Image Retrieval

Rohit Girdhar
The Robotics Institute
Carnegie Mellon University
rgirdhar@andrew.cmu.edu

ABSTRACT

Image retrieval is an important and well studied problem in computer vision. In this work, I consider the problem making deep features based image retrieval scalable to multiple millions of images. To that end, I propose a scalable framework for retrieval by hashing the deep features to low dimensional binary hashcodes. I implement and critically analyze a recent quantization algorithm, Iterative Quantization (ITQ) [8], that has been shown to perform well for retrieval on several datasets. More importantly, my implementation is distributed and runs on top of SPARK, hence making it possible to scale to virtually any size datasets, without being limited by the memory and processing capabilities of a single machine. Using the distributed implementation, I show 13× faster training of ITQ when using a 20 node cluster (as compared to a un-vectorized single node implementation). I finally show a brute-force and hashing hybrid approach that performs better than just hashing, at a minor overhead on the computational cost.

1. INTRODUCTION AND RELATED WORK

Image retrieval is the problem of finding similar images from a corpus of images given a query image. Many existing approaches in this field use local descriptors in the Bag of Words paradigm [16, 14]. Combined with inverted file indexes, this is a highly scalable approach to retrieval. Other approaches use an aggregated representation for images by using encoding schemes such as Fisher kernels or VLAD [13, 4]. Such schemes allow for compressed representation for images, which enables them to scale to even larger datasets.

Continuing on the use of aggregated image representations, recently, there has been a lot of interest in using activations from deep neural networks as a generic global image representation. Apart from a variety of recognition tasks, this has also been shown to perform very well in image retrieval [15]. However deep features are typically very high dimensional ($10^4 D$), making it infeasible to store all the features in the memory of a single computer for fast retrieval (1M images requires 80G of memory for features). This makes it hard to scale it up to large number of images.

The standard practice in such a case is to reduce the dimensionality of the features and binarize them. One approach to do that is LSH [7], which projects the data onto randomly selected hyperplanes and sets each dimension as 0 or 1 depending on which side of the hyperplane it falls on. This has been shown to perform really well for a variety of tasks, especially on metric spaces such as Euclidean as well as on non-metric spaces, like cosine. However, as my experiments also show, recent approaches like ITQ [8] and Spectral Hashing (SH) [20] that learn the optimal hyperplanes from the data to hash instead of picking at random, tend to perform much better than LSH.

There has also been some other work in approximate nearest neighbor approaches, as well as in distributing them over computing clusters. Some general work on fast k-nn graph construction include [5] and [21], where the latter improves on the former by making it applicable to non-euclidean spaces as well. The main idea in these works is to segregate the data into smaller clusters using a cheap method like LSH, and then compute a complete graph over the smaller subgraphs. There is also some similar work focused towards vision related applications, such as [11] in which they discuss approximate NN approaches like FLANN, and propose a very simple distributed architecture for the same by running parts of search on different machines. My work is complementary to these as I focus on distributing the learning of the hash function, instead of the nearest-neighbor search. One can use any of the above approaches to compute a complete k-NN graph using the learnt binary representation from this work. Another work [19] proposes a method for scalable k-NN graph construction for keypoint descriptors, which is also different from this work as I focus on global features.

In this work, I develop a distributed hashing approach, Distributed ITQ, that can use a large dataset to learn the optimal transformation for hashing the features into a lower dimensional binary representation. I implement it over spark and show extensive experiments showing correctness and scalability of the implemented approach. I also show the effect of changing various parameters on the computation time. Finally, I propose a hybrid retrieval approach, using hashes to prune the search space and finally re-ranking the top retrievals using the actual raw features to get significantly better performance accuracy. I also discuss a scalable implementation for the same when running on a single computer.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

2. DEEP NETWORKS AND FEATURES: REVIEW

Recently, there has been a resurgence of deep convolutional neural networks (CNN) in almost all sub-fields of computer vision and especially in recognition. A deep CNN is essentially a end-to-end system that is trained directly using data and labels, and all the intermediate representations are learnt automatically (instead of hand-tuning specific filters, as had been the trend in computer vision before). One landmark paper in this field was [10], which introduced a standard network architecture (AlexNet) for ImageNet 1000-class image classification, which is still very popular in the field for a variety of tasks.

Apart from training deep networks for a diverse set of tasks, researchers have also found great use for the layer activations extracted from a deep network as a generic feature representation for images. A recent work [15] used the features from AlexNet trained for ImageNet [6] classification and showed impressive performance on a variety of tasks, including image retrieval on multiple standard datasets such as Oxford5K, Paris6K and so on. Hence, distance between deep features over a standard space like cosine or euclidean works reasonably well or better than traditional bag of words like approaches to image retrieval. In this work, I use features from fc7 layer of AlexNet trained on ImageNet as the standard feature representation and euclidean distance as the distance metric.

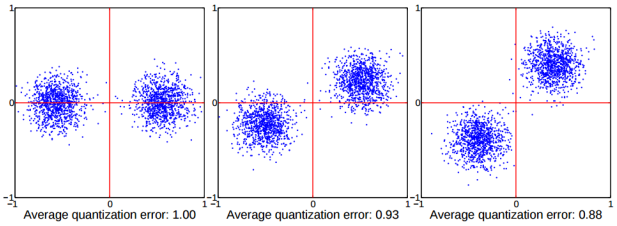
3. ITERATIVE QUANTIZATION: REVIEW

Iterative Quantization [8] is an unsupervised approach to learn similarity preserving optimal binary hash codes. It starts with centered, PCA-projected data and formulates the problem as learning the binary code that directly minimizes the quantization error of mapping the data to the vertices of a zero-centered binary hypercube. They start with a random orthogonal transformation over the PCA projected data (which itself does a good job of balancing the variance along PCA dimensions). Next, they use an EM style minimization approach to refine the initial orthogonal transformation to minimize the quantization error. This idea is better explained by Figure 1, taken from their paper. With the optimal rotation computed using ITQ in Figure 1(c), the clusters of data are separated such that each point in one cluster can be represented as (-1,-1), and other as (1,1).

Next, I explain the problem formulation for binary quantization and the optimization approach. Let X be the $n \times d$ zero-centered data (mean subtracted deep features, in our case), where n is the number of features and d is dimension of each. Let W be the $d \times c$ optimal projection matrix (where c is the final hash code size), and final binary hash code can be computed as $V = XW$ and $B = sgn(V)$ (B is the final hashcode). It can be shown that if W is an optimal solution, then so is WR, for any orthogonal $c \times c$ matrix R (proof in the paper, [8]). So we need to find the orthogonal transformation (R) for the projected data (V) so as to minimize the quantization loss: $||B - VR||_F^2$, where F denotes the Frobenius norm.

The objective can be optimized using an EM style algorithm I describe next. The details on deriving this algorithm can be found in [8] and I skip it here for brevity. The algorithm is described as:

$\mu = mean(X)$



(a) PCA aligned. (b) Random Rotation. (c) Optimized Rotation.

Figure 1: This figure taken from [8] explains the difference in quantizing 2D data by just PCA projection, transforming PCA projected data by a random orthogonal rotation and the same rotated by a rotation computed using ITQ. The square around the data denotes the hypercube in this case, and the closest corner of this hypercube defines the binary representation for a data point.

$X \leftarrow X - \mu$
$W \leftarrow \text{PCA\_projection}(X)$
$R \leftarrow \text{Gaussian\_Random}(c, c)$
$i \leftarrow 0$
for $i < N$ **do**
  // Fix R and update B
  $B \leftarrow sgn(VR)$
  // Fix B and update R
  $S\Omega\hat{S}^T = svd(B^T V)$
  $R \leftarrow \hat{S}S^T$
  $i = i + 1$
end for

The number of iterations (N) can either be defined using a termination criterion (change in R) or, as the authors show, can just be set to a fixed number, 50 for most experiments, as that performs well enough. Finally, for a test feature x, we can compute the binary representation as

$$b = sgn\left((x - \mu) \times W \times R\right)$$

4. DISTRIBUTED ITERATIVE QUANTIZATION

To distribute the ITQ learning computation, one of the most important thing to note is that the vectorized notation I used to describe the algorithm above is not suitable for parallelization. In fact, it has been designed with respect to BLAS operation, and hence expected to run on a single machine. Hence, operations such as a transpose of a large matrix, that exist in the above notation, are not quite efficient in a distributed system (in fact Spark does not even have an API for that).

I implemented this system in Spark because of its ability to handle iterative algorithms better than other distributed systems like Hadoop. I did this implementation in the Python interface for Spark using some data structures and primitives from MLLib. Since this interface is still newer as compared to Scala or JAVA, it exposes fewer features than the other interfaces. For instance, there is no primitive for matrix multiplication, and I had to decompose and distribute these operations manually.

Next, I will explain the way I implemented some of the

operations for this task. The python code for complete implementation (that was also used for the experiments in the later section) can be found on Github.[1] I make the following assumptions in my implementation:

1. Each feature vector individually is small enough to fit in a single computer's memory. In case of deep features, the feature could be a max of 10^4 dimensional, which is 80KB and can easily fit in memory of modern computers. Hence, I represent the matrix of features, X, as a Spark MLLib `RowMatrix` RDD, where rows of this matrix are distributed across nodes

2. The bit length of the hashed feature (c) is a small number, typically less than few thousands. This is because I consider the rotation matrix ($c \times c$) as a local matrix that is mapped to different nodes for operations. Hence, this must fit in memory of a single machine and be light enough to transport across the network on every iteration.

One of the operations in the above algorithm is to compute VR. Here V is a $n \times c$ dimensional matrix, while R is a $c \times c$ matrix. As per the previous assumption, R is a small matrix, and I essentially map it to all the nodes containing rows of V and multiply each row with R. Another important operation is the computation of $B^T V$. Here both $B_{n\times c}$ and $V_{n\times c}$ are very tall matrices. As I mentioned earlier, there is no primitive in spark to transpose a matrix, and the matrix multiplication API is also not exposed in python. Hence, I use the following representation:

$$(b_1^T, b_2^T, \cdots, b_n^T) \times \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \sum_{i=1}^{n} b_i^T v_i$$

where b_i and v_i are rows of the corresponding matrices, each of which are d dimensional (i.e. same as feature's dimension), hence small enough for operation on a single computer. Hence, the above translates into a simple map and reduce operation. Since the matrices are represented as MLLib RowMatrix, I also needed to explicitly keep track of the row numbers to correctly multiply corresponding rows.

5. COMPLETE RETRIEVAL SYSTEM

As we have seen, storing and using the binary hashed features lead to significant savings in storage (both disk and memory) and computation requirements. However, it also leads to somewhat worse performance (accuracy-wise) as compared to using the raw features. One of the main reasons we avoid using raw features is that it is impossible to load them into the memory of a single computer to perform the matching. However, since disks these days have much higher capacities, they can be easily stored and operated on the disk. This, however, would be almost prohibitively slow as most retrieval applications require real-time performance.

Hence, one way to get the best of both worlds is to use the binary hash based matching a way to prune the search space for full raw feature based matching. I call this method 'Re-ranking', as we use the top-n matches from the cheap binary matching and re-rank them using the raw features. As I show in Section 6.7, this leads to significant improvements over just using the binary features.

However, this raises the question of how to efficiently perform this re-ranking step. Since the features need to retrieved from the disk at random and re-ranked, we need a smart way to store them. This would be straightforward if the features were all the same size. For example, if all features were the same dimension, one could simply store them as one large binary file and compute an offset to jump to the specific features we need for re-ranking. However, this would mean storing the features in a raw, uncompressed form. Especially with deep features, which tend to be highly sparse, this ends up being a very inefficient way of storing them.

Hence, one would ideally like to use some sort of compression while storing these features on the disk. Though there are a lot of sparse encoding methods, I used a simple off-the-shelf compression system to compress the features for storage. Specifically, I used the open-source library `zlib` [3] that can compress any given string in a loss-less manner. For the deep features, I serialized it into a string and used zlib to get a compressed version of the same. Overall, it lead to about a $3\times$ reduction in the disk requirement.

However, now all the compressed feature representations are no longer the same size, as different levels of sparsity can give slightly different size representations. Hence, using the simple binary file storage is no longer an option, as a fixed offset for random access is no longer defined. So we need a fast disk based (key, value) data structure to store the features. I use a read-optimized B+ tree implementation for the same. Specifically, I used LMDB [2] which is highly optimized for reads as compared to other popular B+ tree implementations like Google LevelDB [1] or BerkeleyDB [12]. It uses memory mapping (`mmap`) underneath, which automatically caches feature into a memory map and leads to very fast reads for popular/recently accessed features.

6. EXPERIMENTS AND RESULTS

In the following sections, I describe the various experiments I performed on 2 different standard datasets to evaluate the correctness and scalability of the implemented approach.

6.1 Datasets

I evaluated my system on 2 standard datasets: CIFAR10 and Holidays1M.

CIFAR10 [9] is a classification dataset with 50000 training images and 10000 test images, each of which is classified into 10 classes. The images themselves are taken the TINY images datasets [18] and hence are of 32×32 resolution. This is a small dataset by today's notion of 'Big-Data' in computer vision, and I mainly use it because (a) it is a standard dataset used in most nearest neighbor research works, including [8], and (b) it allows comparison with the brute force matching, which might be simply impossible (given my resources) on a larger dataset.

To show the scalability of my approach, however, CIFAR10 is inadequate. Hence, I also experimented with Holidays1M dataset. This dataset contains 1419 labeled images and 1 million distractor images from Flickr. The images are higher quality, with average resolution around 300×500px. I show all the scalability analysis on this dataset.

6.2 Retrieval Quality using ITQ

[1] http://github.com/rohitgirdhar/BigITQ

Table 1: Classification accuracy of the top-500 retrievals using the hashed features and brute force on the CIFAR10 dataset. Note I use a 256bit ITQ or LSH representation.

Method	Retrieval Classification Accuracy	Run Time
Deep Features (LSH)	24.46	8m13s
Deep Features (ITQ)	25.71	7m56s
Deep Features (ITQ) (Using provided code)	25.74	8m10s
Deep Features (Brute Force)	25.40	48m43s

Table 2: Classification accuracy of the top-500 retrievals using 256bit LSH and ITQ as reported in [8] for CIFAR10.

Method	Retrieval Classification Accuracy
GIST (LSH)	26.0
GIST (ITQ)	28.0

To ascertain the quality and correctness of my implementation, I first compare the quality of retrievals using my approach with a baseline LSH implementation, and the MATLAB implementation provided by the original authors.

Table 1 shows the retrieval classification accuracy for different methods and implementations on the CIFAR10 dataset. The metric, retrieval classification accuracy is defined as percentage of top-500 retrievals that are of the same class as query image. I compute these scores over all the 10K test images (note that the ITQ model is not trained on these images). I use this metric as this is one of the metrics used in [8]. I compare the results I obtain for my implementation for ITQ with the provided MATLAB implementation and see almost exactly same performance, hence showing that my implementation gives same results. The slight difference can be attributed to the fact that this algorithm is initialized randomly and hence might converge to slightly different minimas. The runtime in this table corresponds to the total testing run-time for all the 10K test images, on a single core 3.6GHz machine with 24GB of RAM. This does not include the time for training the ITQ model or generating the corpus hash codes for LSH or ITQ. Also note that all these numbers are reported for a 256bit representation.

Table 2 shows the same metric (retrieval classification accuracy), but as reported in the paper [8]. The paper uses a different feature representation, GIST [17], which gives a 320 dimensional feature representation. I observed that using deep features performs worse than GIST, contrary to the belief in Section 2. One reason for this anomaly could be the fact that CIFAR10 has 32×32 small images, which are very different from the 256×256 images from ImageNet that the deep network is trained on (I use the deep network without any finetuning). GIST, on the other hand, is designed for the Tiny Images dataset [18], of which CIFAR10 is a subset. Hence, this discrepancy is not completely surprising. In any case, the performance of deep features is comparable to GIST, and the improvement of ITQ over LSH is almost

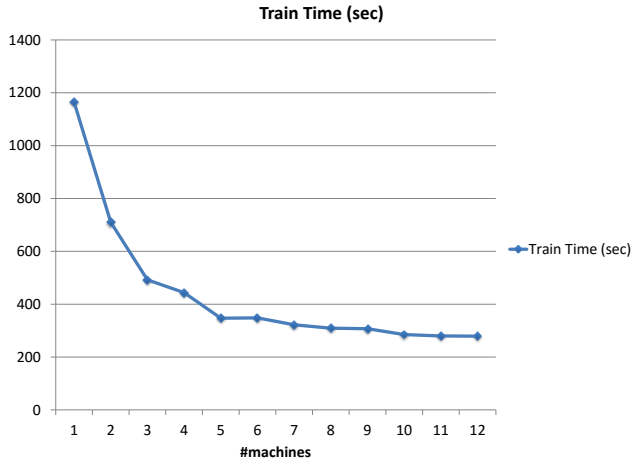


Figure 2: Variation in time to run 50 iterations of ITQ on CIFAR10 using varying number of executors.

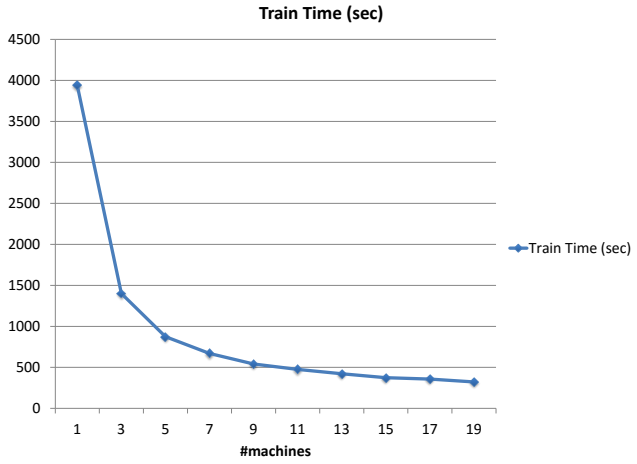


Figure 3: Variation in time to run 10 iterations of ITQ on Holidays1M using varying number of executors.

exactly the same as reported in the paper. Also, note that the testing paradigm in the paper is different from what I follow. The paper uses 5 folds of random 1000 train images as test, whereas I use the complete 10K held-out test set for testing.

6.3 Distributed ITQ performance with Number of Executors

Now, I analyze the scalability of my ITQ implementation with number of machines. Figure 2 shows the time to run 50 iterations of ITQ on different number of executors for the CIFAR10 dataset. Each executor in this case was a 2-core machine with 10GB RAM. Note that this time does not include the time to compute initial centering of the data and PCA projection. As expected, the train time falls linearly at first, and plateaus down eventually. This happens because the additional computing power provided by additional machines no longer trade-offs well with the additional network

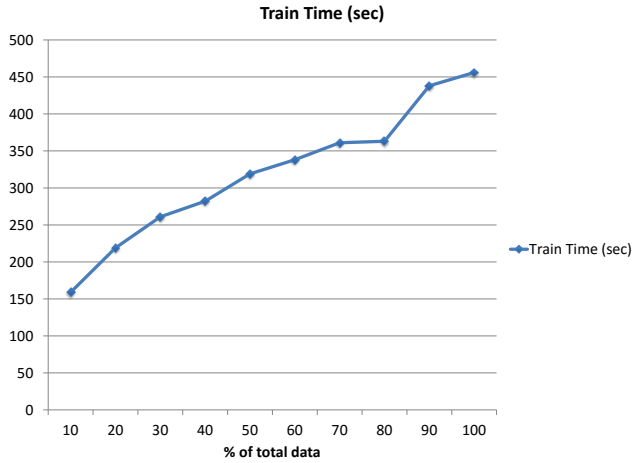


Figure 4: Variation in time to run 50 iterations of ITQ on Holidays1M for varying amounts of randomly subsampled data.

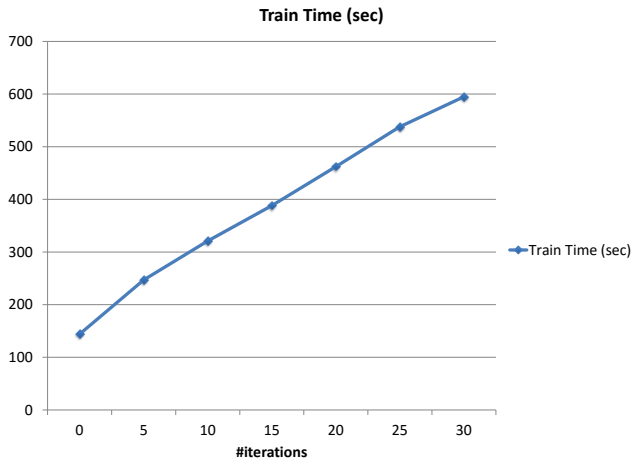


Figure 5: Variation in time to run different number iterations of ITQ on Holidays1M.

overheads.

However, to show that my implementation can actually make use of more nodes, I also performed the same experiment on Holidays1M. Figure 3 shows the corresponding results for this dataset. For this dataset, I run 10 ITQ iterations on 4-core machines each with 80GB RAM. Hence, in this case we observe an advantage on adding more executors till as many as 20. This shows that this implementation can potentially make use even more nodes for even larger datasets.

6.4 Distributed ITQ performance with Amount of Data

Next, I experimented with changing the amount of data used to train ITQ. I randomly selected a subset of images to train from the Holidays1M dataset. I recorded the amount of time to run 20 iterations using 20 executors, each with 4 cores and 80GB RAM. Figure 4 shows this variation. I observed that the increase was almost linear with the amount of data, which is as one would expect. This also indicates that the system would be able to scale in a similar fashion to learn from even larger datasets.

6.5 Distributed ITQ performance with Number of Iterations

Next, I evaluated the effect of number of iterations on the total time taken for ITQ. This would help in determining how the computational expense is distributed across iterations. I performed an experiment on Holidays1M dataset with 20 executors, each with 4core, 80GB RAM, running ITQ for variable number of iterations. Figure 5 shows the results. Hence, the time varies linearly with number of iterations, showing that all subsequent iterations take nearly the same amount of time. One interesting observation here is that the first iteration typically takes much longer than the subsequent iterations. This could be attributed to the initial shuffle required to distribute the data to different nodes, and after that the only data transfer required is the mapping of R matrix to all the nodes.

6.6 Distributed ITQ performance with Hash Code Size

In this experiment, I compared the time taken to run the complete ITQ process with varying the number of bits in the final hash code expected. Note that as compared to previous experiments, I re-ran the PCA computations too, because those can be affected by the number of dimensions of PCA we need to preserve.

Figure 6 shows the total time taken and time per iteration for CIFIAR10 dataset, as I vary the hash code bit length. Figure 7 shows average per-iteration time for the same. I observed that the both total time and average per-iteration varied almost linearly with the number of bits (note the graph is log-scale). This can be explained by the fact that bitlength defines the size of the rotation matrix that is mapped to all the nodes and multiplied with each feature. Hence, it linearly increases the communication cost and processing cost to multiply with each feature. The PCA computation presumably is not effected by the number of bits, as that only specifies the number of dimensions to preserve for further processing.

6.7 Results of the Complete Hybrid System

Finally, I verify the importance of re-ranking retrievals using the actual deep features to get better matches. In practice, I re-rank the top-100 closest matches in hamming distance space over hash codes to get the final retrieval ranklist. Table 3 compares the retrieval performance with or without re-ranking. This analysis is done over the Holidays1M dataset, and the metric used is mean precision at k, defined as ratio of top-k retrievals that match to query. I analyzed it for $k = 1, 3, 5$, because beyond that the mP values are very small (since there are only a few matching images to each query, most of the corpus is full of distractors). As expected, re-ranking significantly improves the final retrievals.

Figure 8 shows some qualitative results of the final retrieval system. It shows two rows per image, one without re-ranking and the other with. Again, re-ranking preserves or improves the retrievals. Overall, both with and without re-ranking perform reasonably well, and one can choose to

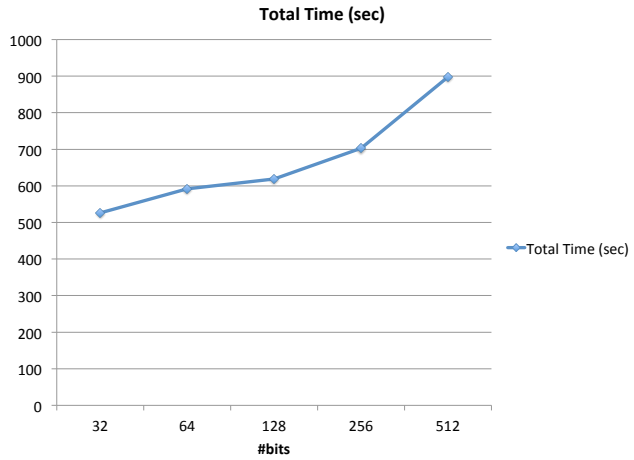


Figure 6: Variation in total time taken for ITQ (including PCA) with different bitlength of hash code to be learnt.

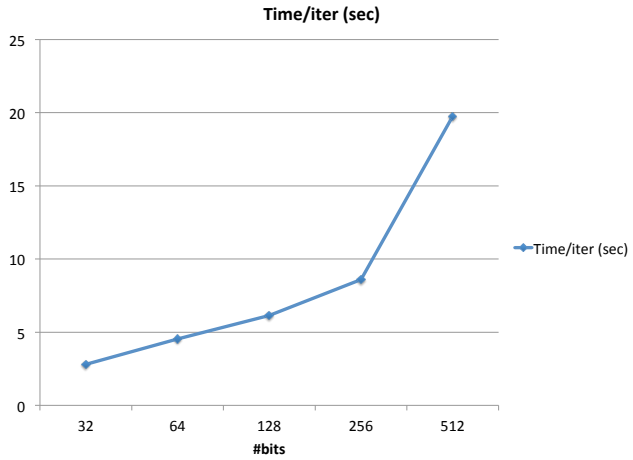


Figure 7: Variation in average time taken per iteration of ITQ with different bitlength of hash code to be learnt.

Table 3: This compares the performance of the system on Holidays1M dataset for retrieving good matches to the top of the list. mPk is the mean precision at k

Method	mP1	mP3	mP5
Without Re-rank	0.422	0.018	0.000
With Re-rank	**0.586**	**0.042**	**0.008**

go with either depending on computational budget.

7. CONCLUSION

In this work, I present a scalable approach to perform instance level retrieval over large datasets using a global feature representation. I do that by hashing the global features (deep features, in this case) into binary hashcodes that are cheaper storage and computation wise. I developed (and release code as open-source) for a distributed iterative quantization framework running over spark that can use multiple machines to scale and speed-up the learning of these optimal hash codes. I also show extensive analysis of this system on two datasets, including effect on its performance with respect to a number of parameters. Finally I show how a hybrid approach out-performs the approach of just using deep features alone, and also propose and implement a scalable and fast way to do it.

8. FUTURE WORK

One extension of this work that would be very helpful is to make the per-node computations during ITQ to be vectorized. Currently I operate on each feature separately, given my limited understanding of spark. However aggregating all the features in the same machine into a single matrix and using BLAS operations to operate can very significantly improve the performance further.

Also, the final retrieval system I designed runs out of a single computer. Another extension to this work could be a distributed retrieval system, that can use multiple machines to compute the nearest neighbors. This can be especially helpful for the re-ranking stage, if one could distribute the features in such a way that features matching to a given feature are distributed across multiple machines and each of the machines can do part of the re-ranking to speed it up.

9. CODE

The code for my Distributed ITQ implementation is available at http://github.com/rohitgirdhar/BigITQ/.

10. REFERENCES

[1] Leveldb leveldb is a light-weight, single-purpose library for persistence with bindings to many platforms. http://leveldb.org/.

[2] lmdb: Symas lightning memory-mapped database (lmdb). http://symas.com/mdb/.

[3] zlib: A massively spiffy yet delicately unobtrusive compression library. http://www.zlib.net/.

[4] R. Arandjelović and A. Zisserman. All about VLAD. In *CVPR*, 2013.

[5] Jie Chen, Haw-ren Fang, and Yousef Saad. Fast approximate k nn graph construction for high dimensional data via recursive lanczos bisection. *The Journal of Machine Learning Research*, 10:1989–2012, 2009.

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. *CVPR*, 2009.

[7] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[8] Yunchao Gong and Svetlana Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. In *CVPR*, 2011.

[9] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*. 2012.

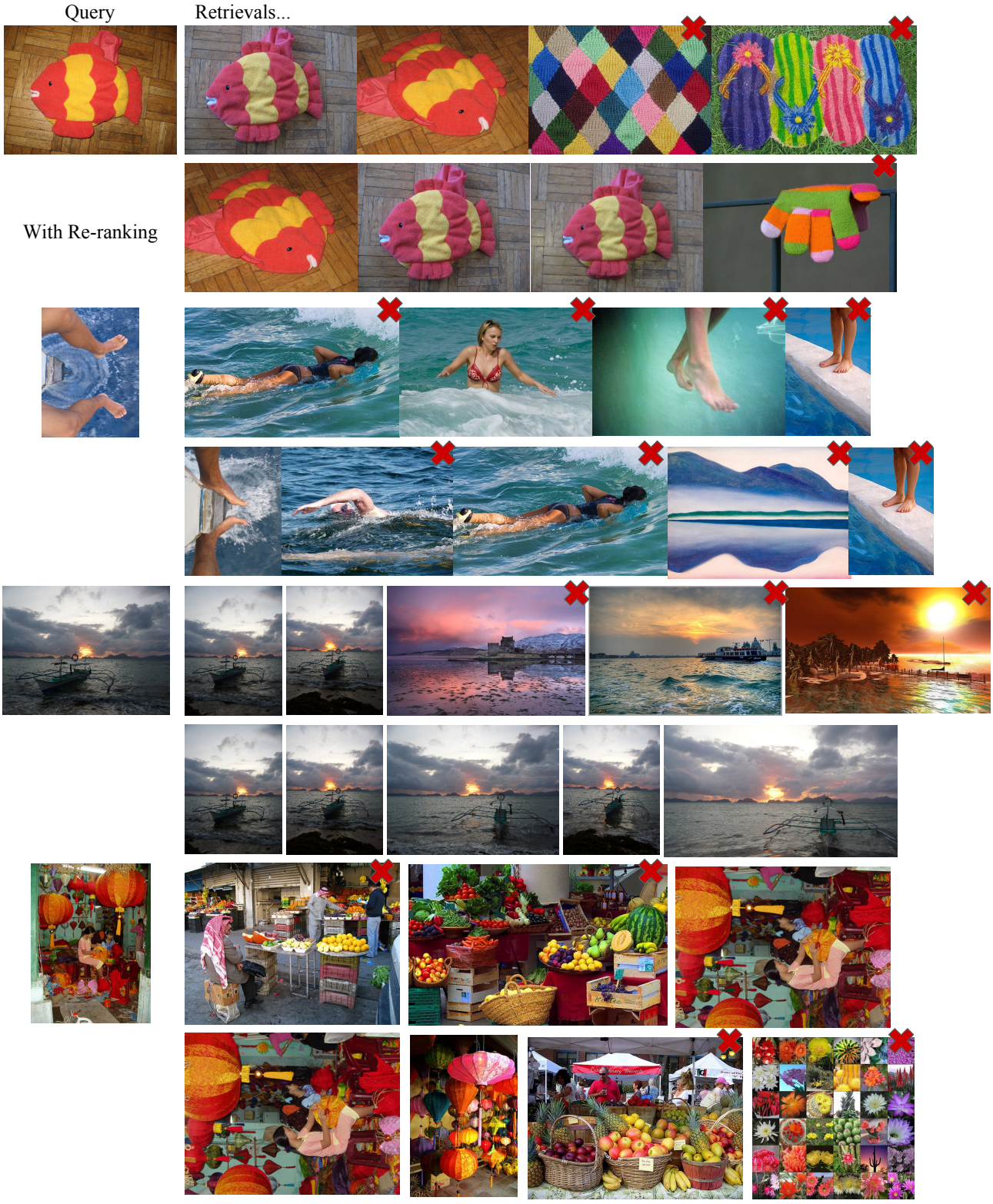


Figure 8: Some qualitative results of retrieval. For each image I show two rows, first using only hamming distance on ITQ hashed space, and second with re-ranking top-100 matches using actual deep features. As expected, re-ranking improves the performance. Crosses on top-right of an image specifies that it is an incorrect match as per supplied ground truth.

[11] Marius Muja and David G Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(11):2227–2240, 2014.

[12] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 1999.

[13] F. Perronnin, Yan Liu, J. Sanchez, and H. Poirier. Large-scale image retrieval with compressed fisher vectors. In *CVPR*, 2010.

[14] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *CVPR*, 2007.

[15] Ali Sharif Razavia, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *CVPR Workshops (DeepVision)*. 2014.

[16] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *ICCV*, 2003.

[17] A. Torralba, K. P. Murphy, W. T. Freeman, and M. A. Rubin. Context-based vision system for place and object recognition. In *ICCV*, 2003.

[18] Antonio Torralba, Rob Fergus, and William T Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *PAMI*, 2008.

[19] Jing Wang, Jingdong Wang, Gang Zeng, Zhuowen Tu, Rui Gan, and Shipeng Li. Scalable k-nn graph construction for visual descriptors. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 1106–1113. IEEE, 2012.

[20] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, 2008.

[21] Yan-ming Zhang, Kaizhu Huang, Guanggang Geng, and Cheng-lin Liu. Fast knn graph construction with locality sensitive hashing. In *Machine Learning and Knowledge Discovery in Databases*, pages 660–674. Springer, 2013.